

Article

Revisiting the Inference Time Optimization of TinyML for ARM-Based Microcontrollers

Chan-Kyu Lee ¹, Seung-Ryeol Ohk ¹  and Young-Jin Kim ^{2,*} 

¹ Department of Electrical and Computer Engineering, Ajou University, Suwon 16499, Republic of Korea; changyu.lee.ece@ajou.ac.kr (C.-K.L.); dhrtmdfuf@ajou.ac.kr (S.-R.O.)

² Department of Mobility Engineering, Ajou University, Suwon 16499, Republic of Korea

* Correspondence: youngkim@ajou.ac.kr

Abstract

As AI research advances, various performance optimization techniques have been studied for TinyML models on microcontrollers with very limited resources. In TinyML frameworks such as TFLM (TensorFlow Lite Micro) and NNOM (Neural Network on Microcontrollers), their execution engines mainly depend on CMSIS-NN for inference acceleration, which is an ARM's back-end library to execute optimized kernel functions for performance optimization. But we often find that CMSIS-NN is not invincible for inference time optimization on TinyML. In this paper, we examine TinyML frameworks and their CMSIS-NN libraries and consider how to improve CMSIS-NN in terms of runtime. Then, we propose the D2I technique to reduce the overhead of memory operations that occur while performing the Im2col procedure within the convolution function, which takes most of the inference time in CMSIS-NN. The proposed technique creates a necessary index table, finds the location of the input with the corresponding index, and performs direct operations between filters and inputs. Thus, it can quite mitigate data copy operations in Im2col with a small additional amount of memory compared to Im2col. In extensive experiments using an Arduino nano 33 BLE board with Cortex-M4 and an STM32F746G-DISCO board with Cortex-M7, D2I was found to achieve about 16.3% and 14.5% inference time improvements against the Im2col in TFLM's and NNOM's CMSIS-NNs, respectively, for the SqueezeNet model. And the additional memory usage was shown to be identically 11.52 kB.

Keywords: inference time; optimization; TinyML; framework



Academic Editors: Maria Liz Crespo, Alexander Gegov, Patrick Siarry, Kui Jiang, Xin Xu and Xin Yuan

Received: 22 May 2026

Revised: 30 June 2026

Accepted: 6 July 2026

Published: 8 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

AI is broadly divided into cloud AI, which performs both training and inference on a server, and edge AI, which includes on-device AI, which performs training on a server and inference on a device. Recent hardware advances have led to a significant increase in edge AI models, such as voice recognition and translation on smartphones, real-time retouching of photos, and error detection AI models on IoT devices in smart factories. Especially in microcontroller environments, where computing power and memory resources are very limited and power consumption is small, it is important to properly consider the speed of execution and power consumption when processing AI model computations [1,2].

Tiny Machine Learning (TinyML) is suitable for running AI on microcontrollers, which are small, low-power computing devices. The global TinyML Market size was evaluated at USD 1.47 billion in 2023 and is predicted to become USD 10.80 billion by 2030, with a CAGR of 24.8% from 2024 to 2030 [3]. There are various research efforts to optimize

TinyML performance, mainly in model lightweighting and acceleration research, such as pruning, quantization, and convolution filter compression. However, in addition to model lightweighting, runtime research for efficient use of the model itself is also very important. Therefore, various deep learning frameworks for microcontrollers have been studied.

As shown in Figure 1, the most widely used framework for deep learning in TinyML model lightweighting research is Google's TensorFlow Lite for Microcontrollers (TFLM) [4], which, in particular, uses CMSIS-NN [5] as a technique to optimize neural network computation on ARM-based microcontrollers, including the Cortex-M series. Neural Network on Microcontroller (NNOM) [6], a framework used in conjunction with ARM's backend library CMSIS-NN, is a framework that enables the execution of neural network models on microcontrollers. NNOM is designed to optimize memory usage and computational speed so that deep learning models can run smoothly on microcontrollers with limited resources.

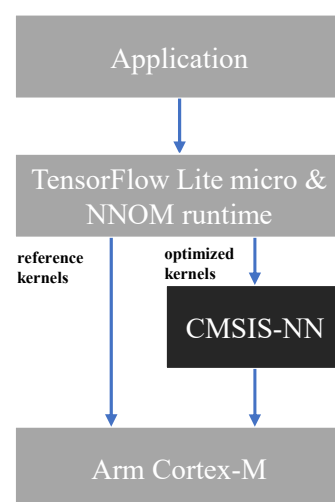


Figure 1. Run-time flows of TFLM and NNOM.

However, the framework used by TinyML relies heavily on ARM's deep learning library, CMSIS-NN, for performance optimization. Figure 2 shows the total inference time, convolution time, and CMSIS-NN library function execution time during convolution for the MNIST model with TFLM and NNOM. In the revised experiments, convolution accounts for approximately 99% of the total inference time in both TFLM and NNOM. In addition, the CMSIS-NN function execution time accounts for approximately 98% of the convolution execution time in both frameworks. These results indicate that CMSIS-NN dominates the convolution workload and accounts for a substantial portion of the overall inference time. Therefore, evaluating the performance of CMSIS-NN is essential for optimizing TinyML execution performance.

Because CMSIS-NN uses SIMD (single instruction, multiple data) to exploit the parallelism of multiple computing units to optimize deep learning inference time on ARM, it is not easy to develop optimization techniques that surpass CMSIS-NN. Among the optimization functions provided by CMSIS-NN, convolution uses the GEMM (General Matrix Multiplication) method as its operation method, which is also called Im2col (Image to column). The Im2col method reduces the amount of computation by converting a multidimensional array input into a one-dimensional array to simplify computation and increase the parallelism of computation by connecting it with SIMD. However, the overhead incurred when converting high-dimensional inputs into one-dimensional arrays can lead to a bottleneck in which the computation speed is fast, but it takes a long time to enter the computation.

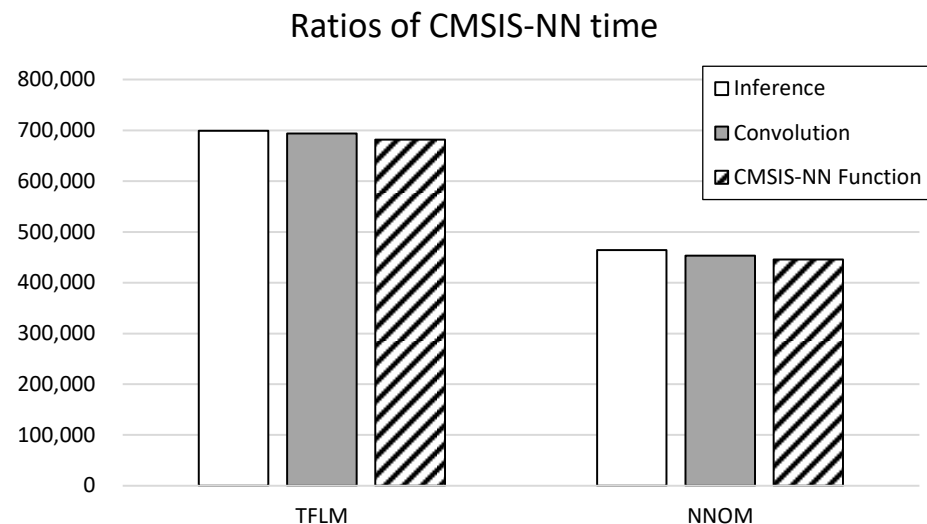


Figure 2. Proportion of the CMSIS-NN execution time against the whole convolution time.

This paper analyzes the deep learning runtimes of TFLM and NNOM and their CMSIS-NN functions for ARM-based microcontrollers, and proposes the D2I (Direct to Image) technique to optimize inference time against Im2col by using an efficient memory access method and enhancing support for SIMD operations.

2. Background and Related Work

2.1. Frameworks for Deep Learning for Microcontrollers

In addition to TFLM and NNOM, Machine Learning Operations (MLOps) refers to a set of practices, tools, and philosophies for automating and optimizing the entire lifecycle of machine learning models, including development, deployment, monitoring, and management [7]. One such research that aims to implement such MLOps in the constrained environment of microcontrollers is Tiny-MLOps [8]. Since the traditional MLOps approach cannot be used due to limited resources, this research aims to develop a framework to effectively manage machine learning models for microcontrollers through specialized tools and methodologies, so that the entire lifecycle of deep learning models can be effectively managed on devices with limited resources.

Other known frameworks for deep learning on microcontrollers include PyTorch Mobile by Meta (formerly Facebook), SNPE by Qualcomm, NCNN by Tencent [9], and the recently developed CubeAI by STM [10]. In this paper, we optimize the execution engine based on TFLM and NNOM, which are the most representative open-source execution engines.

2.2. Research on Model Lightweighting and Acceleration

Model lightweighting techniques can be divided into four categories: parameter pruning and quantization, low-rank factorization, compact convolution filters, and knowledge distillation [11]. Parameter pruning and quantization are the most representative and well-known of the many model lightweighting techniques. Pruning is a technique that removes unnecessary neurons, weights, and network connections to reduce the size of deep learning models and improve their efficiency [12]. This model lightweighting technique allows the model to use less memory and computational resources, allowing the model to run effectively on resource-constrained microcontrollers. Quantization, like pruning, is a technique used for deep learning lightweighting that reduces the size and computation of the model by expressing its weights and activation values of the model in a lower number of bits [13]. Along with pruning, various quantization methods can be used to

reduce memory usage and computational resources while minimizing model performance degradation [14].

Compression of convolutional filters is a way to reduce computational cost and improve performance by designing filters used in convolutional neural networks in a more efficient and lightweight form. Representative methods include Depthwise Separable Convolution, which performs convolutional operations independently for each input channel, and pointwise convolution, which uses 1×1 convolution to create coupling between channels to significantly reduce the amount of computation [15].

On the other hand, Direct Convolution is very simple to implement because it directly multiplies the input data and the filter to produce the output. This convolution has advantages over GEMM due to its memory efficiency and simple implementation, but it is difficult to parallelize the operation, and the computational effort increases significantly with large input data. For TinyML models with relatively small layer sizes, Direct convolution can be competitive with GEMM because the preprocessing overhead of GEMM-based Im2col may outweigh its computational advantage.

Direct convolution on ARM multi-core is one of the optimization techniques in nDirect, which provides new optimizations for data packets and parallelism targeting ARM-based multi-core platforms commonly used in smartphones and HPC systems [16]. Indirect convolution is a technique that aims to provide the existing efficiency of GEMM without the conversion overhead of im2col. It performs the computation using an indirect pointer buffer that points to each start of a pixel without reconstructing the input data [17]. In addition, recent memory-efficient GEMM/convolution methods such as mGEMM [18] reduce the memory overhead of convolution on ARMv8-A/NEON-based mobile processors. However, nDirect and mGEMM mainly target ARMv8-A/NEON multi-core platforms rather than Cortex-M microcontrollers, and applying them to the Arduino Nano 33 BLE Sense requires substantial architecture-specific reimplementation because of the different instruction set, memory hierarchy, and cache structure. In particular, the nRF52840 used in the Arduino Nano 33 BLE Sense provides only a 2 kB instruction cache, which makes it difficult to directly apply complex micro-kernel-based optimizations designed for mobile processors. Therefore, in this paper, we experimentally compare D2I with Direct convolution and SIMD-optimized Direct convolution baselines that can be implemented consistently on the Cortex-M4 target platform.

The difference between nDirect and Indirect convolution is that both focus on optimizing the convolution itself, while D2I, presented in this paper, focuses on optimizing the CMSIS-NN. In particular, Indirect convolution and D2I are similar in that they reduce the data conversion overhead of Im2col, but D2I is different from Indirect convolution because it not only reduces the data conversion overhead of Im2col by using a table indexing the location of input data in the convolution operation of CMSIS-NN, but also performs SIMD operations through the created table better than Im2col.

Knowledge distillation is a technique that uses a large, complex, and high-performance model, called a teacher model, and soft targets, which are probability distributions predicted by the teacher model over the input data, to learn a smaller, simpler, and lighter model than the teacher model, while retaining as much of the teacher model's knowledge as possible [19].

3. Motivation

One of the most popular performance optimization techniques used in TinyML is to use CMSIS-NN, a deep learning back-end library available on ARM. CMSIS-NN uses SIMD, which makes it very difficult to beat parallelism-based optimizations from a computational perspective. However, there has been little exploration of whether CMSIS-NN provides

sufficient optimizations for some preliminary steps, such as data allocation, that lead to parallelism-based computation.

Among the optimization kernels provided by CMSIS-NN, the convolution kernel is traditionally implemented as a GEMM. It performs Im2col, which copies the multidimensional input data into a one-dimensional buffer and imports it as matrix data before entering the computation and performing SIMD. This can significantly reduce the amount of computation by turning operations on multidimensional arrays into simple one-dimensional matrix operations. However, using Im2col does not always result in better performance. Im2col introduces overhead due to the potential for increased memory usage as the input data is expanded and converted to matrix data, as well as additional preprocessing to convert the input data.

Figures 3 and 4 show the computational methods of Im2col and D2I, respectively. In Figure 3, Im2col takes 9 input data, converts them to a one-dimensional array, and performs a SIMD operation on the data and the filter to produce output 59. The input data is copied to a buffer by filter size $\times$ input channel, and the input is read from that buffer to perform the filter and SIMD operations. Because the SIMD operation is repeated by the size of the output, the total number of data copies performed is filter size $\times$ input channel $\times$ output size, which is 6084 total data copies in the example. In Figure 4, D2I first creates an index table based on the input channel and filter size, then uses the created table to read the index value of the table and perform SIMD operations between the input data and the filter corresponding to it. D2I's table contains index values for 9 inputs. The first three values in the table are 0, 1, and 2, which can be used to access input data 4, 1, and 3. The SIMD operation with the filter is performed on all 9 inputs (3×3 matrix), including these inputs, to produce output 59. In the example operation, the table creation is performed 18 times, and the process of reading the table values to get the input data is performed 5048 times. Since this is 1018 operations fewer than Im2col, the execution operations in D2I are reduced due to the reduction in the number of data accesses in memory despite the same number of SIMD operations.

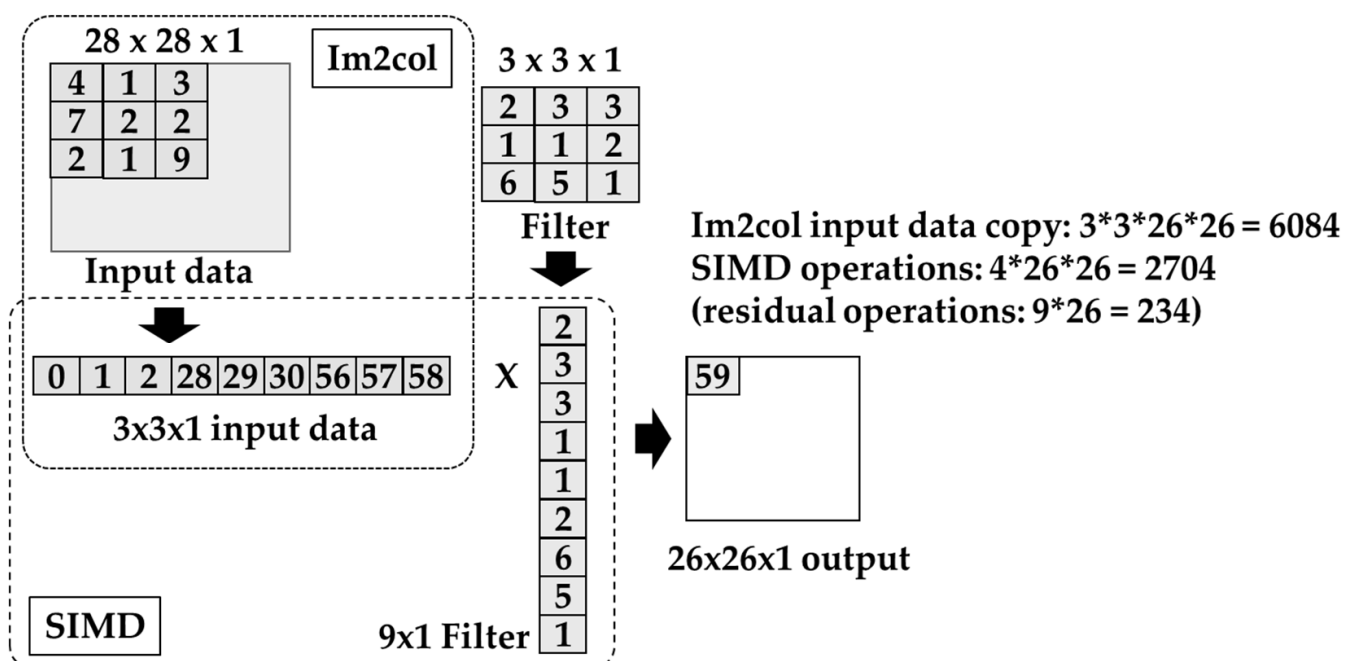


Figure 3. Im2col operations in CMSIS-NN.

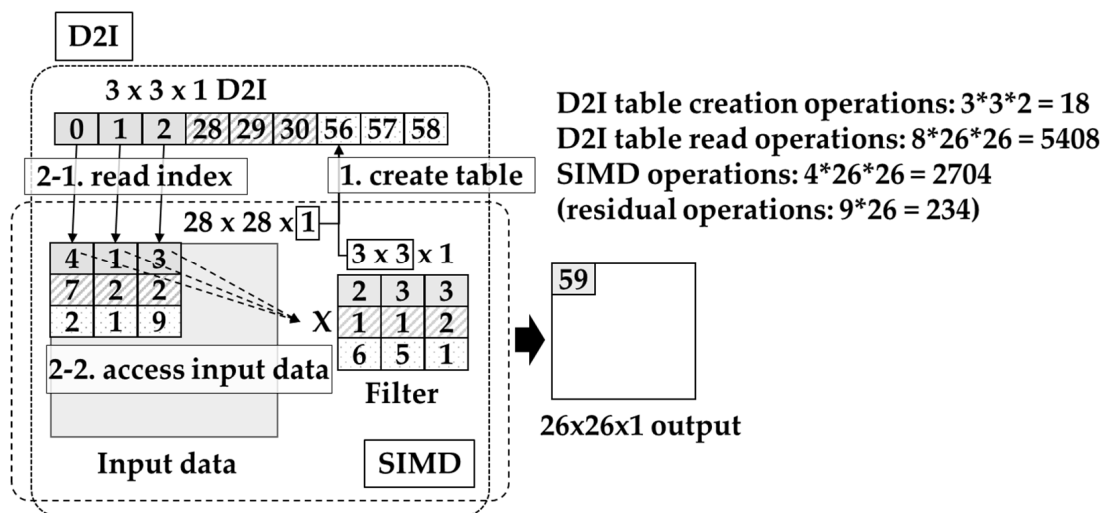


Figure 4. D2I operations.

To generalize this motivation, consider a simplified convolution layer with a single $N \times N$ input feature map, a $k \times k$ filter, a stride of 1, and no padding. The output feature map size is $M \times M$, where $M = N - k + 1$. In an Im2col-based implementation, k^2 input elements are copied or reorganized for each output position, and the copy-related cost can be expressed as $k^2 M^2$. In contrast, D2I removes repeated input-buffer copying and instead uses a precomputed indexing table. The corresponding table-generation and table-access costs can be approximated as $2k^2 + (k^2 - 1)M^2$, where $2k^2$ represents table construction for two adjacent output positions and $k^2 - 1$ reflects the reduced indexing overhead after the base reference is determined. Therefore, the expected gain of D2I over Im2col in this copy/indexing component is

$$\frac{k^2 \times M^2}{k^2 \times 2 + (k^2 - 1) \times M^2}$$

For the common 3×3 convolution case, this ratio approaches $9/8 \approx 1.125$ as M becomes large, indicating an expected improvement of approximately 11.1% in the copy/indexing component. This explains why D2I is effective for the 3×3 convolution layers commonly used in TinyML CNN models, while the relative gain becomes smaller when the computation is dominated by SIMD MAC operations.

4. Proposed Approach

In this paper, we propose a data reference method, D2I, which has values representing the location of input data in the form of a table, and if the table is referenced when performing operations, the input data can be accessed directly without having to find and copy the necessary data. According to TinyML models, D2I checks whether padding is applied to the convolution layer and whether the input channel is a multiple of 4. Then it generates a table differently depending on the result of the check. D2I tackles optimizing the SIMD operations over the table for each result. Algorithms 1–4 describe the table-generation algorithm and SIMD operation algorithm for each check result in detail. In this paper, the input tensor is assumed to be stored in NHWC layout. Therefore, the linear offset of an input element $I[y, x, c]$ is defined as $\text{offl}(y, x, c) = (y \times W + x) \times C + c$, where H and W denote the input height and width, respectively, and C denotes the number of input channels. The D2I table stores these linear offsets or base offsets, depending on whether padding is applied.

Algorithm 1: D2I Convolution

Variables:

C: input channel, H: input height, W: input width

K: number of filters, R: filter height, S: filter width

P: output height, Q: output width

Px: pad height, Py: pad width, St: stride

Layout:

I is stored in NHWC order.

$\text{offl}(y, x, c) = (y \times W + x) \times C + c$

Input:

$I[C \times H \times W]$, Filter: $F[K \times C \times R \times S]$

Table Buffer:

$B[2 \times C \times R \times S]$

Output:

$O[K \times P \times Q]$

```

1:  num_col = C × R × S
2:  if (Px == 0 and Py == 0)
3:      Invoke create D2I table
4:      for p = 0 to P − 1 do
5:          for q = 0 to Q − 1 step 2 do
6:              base = offl(p × St, q × St, 0)
7:              Invoke D2I SIMD computation
8:          end for
9:      end for
10: else
11:     for p = 0 to P − 1 do
12:         for q = 0 to Q − 1 do
13:             Invoke create D2I pad table
14:             if two output columns are buffered
15:                 Invoke D2I SIMD computation
16:             end if
17:         end for
18:     end for
19: end if

```

Algorithm 2: Create D2I table

```

1:  i = 0
2:  num_col = C × R × S
3:  for r = 0 to R − 1 do
4:      for s = 0 to S − 1 do
5:          for c = 0 to C − 1 do
6:              B[i] = offl(r, s, c)
7:              B[i + num_col] = offl(r, s + St, c)
8:              i++
9:          end for
10:     end for
11: end for

```

Algorithm 3: Create D2I pad table

```

1:  i = 0
2:  for r = 0 to R − 1 do
3:      for s = 0 to S − 1 do
4:          ky = St × p − Px + r
5:          kx = St × q − Py + s
6:          if (ky < 0 or ky ≥ H or kx < 0 or kx ≥ W)
7:              B[i] = −1
8:          else
9:              B[i] = offl(ky, kx, 0)
10:         end if
11:         i++
12:     end for
13: end for

```

Algorithm 4: D2I SIMD computation

```

1:  num_col = C × R × S
2:  if (Px == 0 and Py == 0)
3:      for i = 0 to num_col − 1 do
4:          b0[i] = I[B[i] + base]
5:          b1[i] = I[B[i + num_col] + base]
6:      end for
7:  else
8:      Expand pad table B into b0 and b1
9:      // if B[j] == −1, the corresponding input values are filled with 0
10: end if
11: row_count = K/2
12: while (row_count)
13:     col_count = num_col/4
14:     for j = 0 to col_count − 1 do
15:         a01, a02, a11, a12 = read filter data in F
16:         SIMD function(b0, b1, a01, a02, a11, a12)
17:     end for
18:     handle remaining num_col % 4 elements
19:     row_count--
20: end while

```

4.1. Different Table Creation Functions According to Parameters in Layers

D2I essentially skips the input data conversion process performed by Im2col and instead generates a table representing the locations of the input data. At this time, D2I checks whether padding is applied to the layer where convolution is performed. If padding is applied, Algorithm 3 is performed to create a table, considering that padding is applied to the input data. If padding is not applied, Algorithm 2 is performed to create a table.

Without padding, Algorithm 2 creates a reusable D2I table for two adjacent output positions. The table size is $2 \times C \times R \times S$, because the SIMD kernel computes two output columns at a time and each output column requires $C \times R \times S$ input elements. Once this table is created, it can be reused for the remaining output positions by adding the base

offset of the current output position. Therefore, D2I avoids repeatedly reconstructing the Im2col buffer and only updates the base offset for the next output position.

Compared to Im2col, D2I does not repeatedly copy input data into an Im2col buffer. Instead, it creates a reusable offset table and updates only the base offset for each output position. This reduces repeated memory-copy overhead while preserving SIMD-based convolution computation.

If padding is applied, D2I performs Algorithm 3 to create the padding-aware table. In Algorithm 3, D2I first computes the input coordinate corresponding to each output position and filter position as $ky = St \times p - Px + r$ and $kx = St \times q - Py + s$. If the coordinate is outside the input boundary, the table entry is set to -1 . Otherwise, the table stores the base offset $offl(ky, kx, 0) = (ky \times W + kx) \times C$. The channel offset c is added later when the input values are expanded for SIMD computation. Since padding changes the valid input positions depending on the output coordinate, the padding-aware table must be generated for each output position rather than being reused globally. Accessing data directly in SIMD operations through the table created by Algorithm 3 is the same as without padding, but the performance of D2I is worse than the existing method because the table for the next operation needs to be created again and again. To handle padded positions, D2I marks invalid input locations as -1 in the table and expands them into zero values before SIMD computation. As a result, the main SIMD loop can process the packed input buffer uniformly without additional branch instructions for padded entries.

4.2. SIMD Operations Using Tables

Because the way the table is generated depends on whether the convolution layer is padded or not and the input channel values, the optimal SIMD operation must be performed for each table. SIMD operations perform operations in the direction of the channel first, rather than the order in which the operations are performed, i.e., finishing computing on the input data of one channel and moving on to the input data of the next channel. For this reason, if the input channels are multiples of four, knowing only the position of the input data on the first channel is sufficient to compute the position of the next operation data, but if the input channels are not multiples of four, the position of the next operation data cannot be obtained in a regular way because operations on one to three data remain.

Therefore, if no padding is used and the input channel is not a multiple of 4, D2I creates and uses a table for all inputs per channel, so the location of all input data needed to compute one output must be recorded in the table. On the other hand, if the input channels are multiples of 4, D2I can access all the input data by recording the location of the input data for only one channel. Therefore, when the input channels are multiples of 4, D2I can perform much faster operations, not only because it creates fewer tables but also because it references table values much less often in SIMD operations.

When padding is applied to the convolution layer, D2I uses a padding-aware table generation method, which also changes how the input values are prepared for SIMD computation. For each output position and filter position, the table stores -1 when the corresponding input coordinate refers to padded input space. Otherwise, it stores the base input offset of the valid input element. Before SIMD computation, entries marked as -1 are expanded into zero values in the packed input buffer, while valid entries are expanded into the corresponding input values. Therefore, the SIMD kernel can process the packed buffer uniformly without an additional branch inside the main SIMD loop, and padded positions do not contribute to the accumulated result. Although padding requires repeated table generation, this zero-filling process preserves a regular SIMD computation flow and avoids unnecessary contributions from padded input positions.

5. Experiments and Results

We used TFLM v.2.4 with CMSIS-NN v.5.0 as the inference framework. The main target board was the Arduino Nano 33 BLE Sense, which uses an nRF52840 Cortex-M4F processor operating at 64 MHz. Since D2I is optimized for the convolution operations of CMSIS-NN, tests were conducted with the MNIST [20], CIFAR-10 [21], and SqueezeNet [22] models. All evaluated models are lightweight int8-quantized models for CMSIS-NN execution and microcontroller deployment.

All implementations compared in this study were compiled under the same build configuration. The compiler was arm-none-eabi-g++ 7.2.1 from GNU Tools for Arm Embedded Processors 7-2017-q4-major. The optimization level was -Os. The compile target was ARM Cortex-M4 with the Thumb instruction set, fpv4-sp-d16 FPU, and softfp ABI. The main compile-size options were -Os, -ffunction-sections, and -fdata-sections, and the main linker options included -Wl,--gc-sections, and wrapper options for memory-allocation and program-exit functions. Therefore, the compiler and linker settings were controlled consistently across Im2col, Direct convolution, SIMD Direct convolution, and D2I.

The reported inference times were obtained from repeated executions on the target board, and the same measurement procedure was applied to all evaluated models and implementations. As a representative case, the original SqueezeNet_3 D2I measurement consisted of 10 repeated runs, resulting in an average execution time of 316,624.00 us, a standard deviation of 30.34 us, a 95% confidence interval of $316,624.00 \pm 21.70$ us, and a coefficient of variation of 0.00958%. Similar run-to-run stability was observed in the other evaluated models.

Table 1 summarizes the architectures of the evaluated models using input tensor size, feature-extraction blocks, classifier layers, and parameter count. Conv($k \times k$, C) denotes a convolution layer with C output filters, FC-C denotes a fully connected layer with C output units, GAP denotes global average pooling, and Fire(sq,e1,e3) denotes a SqueezeNet fire module with sq squeeze filters and e1/e3 expand filters. All tensors follow the NHWC layout, and all models output a 10-dimensional class-probability vector.

Table 1. Summary of the evaluated model architectures.

Model	Input	Feature Extractor	Classifier	Params
MNIST_light	$1 \times 28 \times 28$	Conv(3×3 , 12), MaxPool	FC10	20,410
MNIST_middle	$1 \times 28 \times 28 \times 1$	Conv(3×3 , 32), Conv(3×3 , 64), MaxPool $\times 2$	FC64, FC10	121,930
MNIST_high	$1 \times 28 \times 28 \times 1$	Conv(3×3 , 32), Conv(3×3 , 64), Conv(3×3 , 64), MaxPool $\times 2$	FC64, FC10	93,322
Cifar-10_light	$1 \times 32 \times 32 \times 3$	Conv(3×3 , 32), MaxPool	FC64, FC10	462,410
Cifar-10_high	$1 \times 32 \times 32 \times 3$	Conv(3×3 , 32), Conv(3×3 , 64), Conv(3×3 , 64), MaxPool $\times 2$	FC64, FC10	122,570
SqueezeNet_1	$1 \times 32 \times 32 \times 3$	Conv(3×3 , 16), Fire(8, 16, 16) $\times 2$, Fire(16, 32, 32) $\times 2$, MaxPool $\times 2$	Conv(1×1 , 10), GAP	16,058
SqueezeNet_2	$1 \times 32 \times 32 \times 3$	Conv(3×3 , 96), Fire(16, 64, 64) $\times 2$, Fire(32, 128, 128) $\times 2$, Fire(48, 192, 192), MaxPool $\times 3$	Conv(1×1 , 10), GAP	230,554
SqueezeNet_3	$1 \times 32 \times 32 \times 3$	Conv(3×3 , 64), Fire(16, 32, 32) $\times 2$, Fire(32, 64, 64) $\times 2$, Fire(48, 96, 96) $\times 2$, Fire(64, 128, 128) $\times 2$, MaxPool $\times 3$	Conv(1×1 , 10), GAP	365,386

To further verify the performance improvement of D2I in other frameworks and board environments, experiments were conducted on NNOM v.0.4.3 using CMSIS-NN with D2I for SqueezeNet_3 on an STM32F746G-DISCO board equipped with a Cortex-M7 processor operating at 216 MHz.

Table 2 reports the inference-time reduction and memory usage of D2I. The “Number of conv2d” column indicates the number of convolution layers to which D2I was applied, not the total number of convolution operators in the model.

Table 2. Inference time enhancement and memory usage for various models.

Model	Number of D2I-Applied Conv2d	Padding	Reduced Execution Time (%)	Memory Usage (kB)
MNIST_light	1	X	19.5	0.032
MNIST_middle	2	X	15.1	1.184
MNIST_high	3	X	16.5	3.488
Cifar-10_light	1	X	2.8	0.096
Cifar-10_high	3	X	15.3	3.552
SqueezeNet_1	5	O	7.7	5.208
SqueezeNet_2	6	O	9.4	5.496
SqueezeNet_3	9	O	16.3	11.52

5.1. Inference Time and Accuracy for the MNIST Model

In Figure 5, we show how the performance improvement of D2I varies with the number of filters for the MNIST model. The proportion of data copying and SIMD operations caused by performing Im2col in the total execution time is sensitive to parameters such as the input size of the convolution layer and the number of filters. In particular, increasing the number of filters can significantly affect the performance of D2I at reducing data copying, because it only increases SIMD operations and not data copying. As an experiment to verify this, we measured the execution time of Im2col and D2I by changing only the number of filters in the layer in the MNIST model with only one convolutional layer (equivalent to MNIST_light in Table 2). In Figure 5, we can see that the performance of D2I decreases from 52.6%, 41.8%, 34.8%, 30%, and 19.5% as the number of filters increases from 3 to 24, because the increase in the number of filters only increases the SIMD operations. However, as the number of filters decreases, the accuracy of MNIST does not drop significantly from a maximum of 98.1% to a minimum of 95%, but the performance of D2I increases significantly from 19.5% to 52.6%, so if the increase in D2I performance is much larger than the decrease in accuracy due to the decrease in the number of filters, it is significant to improve the performance of D2I by adjusting the number of filters in some convolution layers.

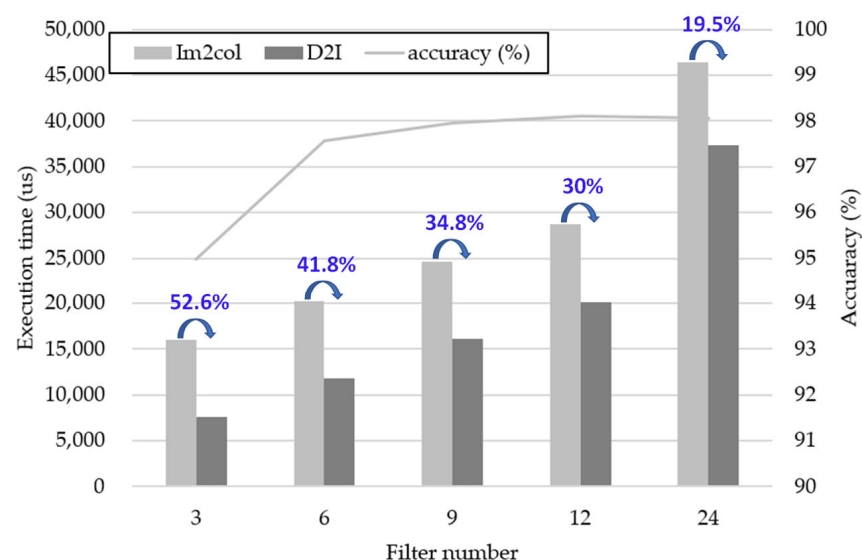


Figure 5. Inference time and accuracy of Im2col and D2I for the MNIST model with different filters.

To analyze this trend in more detail, we compared the number of data-copy operations, SIMD operations, and inference times of Im2col and D2I for each MNIST_light model with different filter counts. As shown in Figure 6, the number of data-copy operations remains constant regardless of the number of filters. Im2col performs 6084 data-copy operations, whereas D2I performs only 347, reducing data-copy operations by approximately 94.3%. This large reduction explains why D2I achieves a shorter inference time than Im2col for all filter counts.

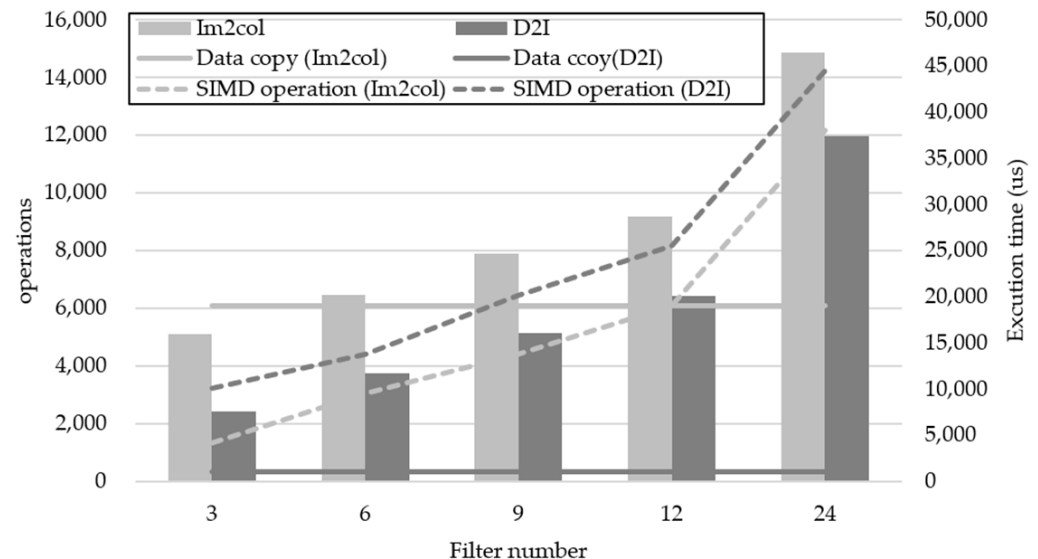


Figure 6. Inference time and operations of Im2col and D2I for MNIST models with different filters.

However, the number of SIMD operations increases as the number of filters increases in both methods. D2I performs more SIMD operations than Im2col because of the additional table-based input referencing process. As the filter count increases from 3 to 24, the inference time of Im2col increases from 15,978 us to 46,424 us, while that of D2I increases from 7566 us to 37,378 us. Therefore, D2I remains faster than Im2col, but its relative performance gain decreases as SIMD operations become a larger portion of the total execution time while the data-copy reduction remains constant.

5.2. Inference Time and Accuracy for Various Models

The D2I proposed in this paper can perform operations more efficiently when the input channel count of a convolution layer is a multiple of 4. This condition is important for efficient SIMD packing on Cortex-M processors. The condition refers to the channel dimension of each convolution layer, not to the spatial input size. D2I can still be applied when the input channel count is not a multiple of 4, but SIMD packing and table referencing become less regular, which can reduce the performance gain. In the first convolution layers of MNIST and CIFAR-10 models, the input channel counts are 1 and 3, respectively. D2I is still applicable to these layers, but it is generally more effective in later convolution layers where the channel dimension is aligned to four elements, such as 16, 32, 64, or 128 channels. To evaluate this behavior, we conducted experiments on several models with different numbers of convolution layers. The inference times of Im2col and D2I for these models are shown in Figure 7.

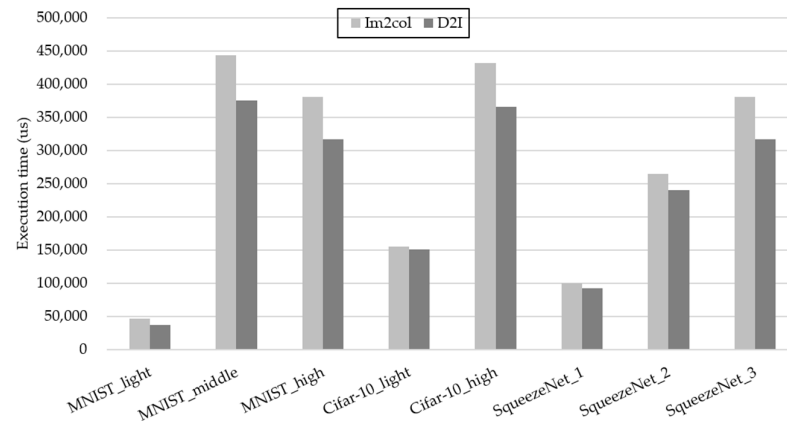


Figure 7. Inference time of Im2col and D2I for various models.

For MNIST, we used MNIST_light, MNIST_middle, and MNIST_high models with one, two, and three convolution layers, and for Cifar-10, we used Cifar-10_light and Cifar-10_high models with one and three convolution layers. From Figure 7 and Table 2, for MNIST_middle and MNIST_high, the D2I gain in the first convolution layer is smaller because the first layer has a single input channel and uses more filters than MNIST_light. In contrast, the later convolution layers have input channel counts aligned to four elements, so SIMD packing and table referencing are more efficient. As a result, MNIST_middle and MNIST_high achieve overall execution-time reductions of 15.1% and 16.5%, respectively.

For the convolution layer after the first, the number of operations increases significantly because the number of input channels and filters is much larger than the first layer, but the table creation and table access when the input channel is a multiple of 4 are significantly reduced compared to D2I when the input channel is not a multiple of 4, which prevents the performance of D2I from decreasing due to the increase in operations. CIFAR-10 shows a similar trend. Cifar-10_light has a single convolution layer with three input channels and 32 filters. Although D2I is still applied to this layer, its performance gain is limited to 2.8% because the input channel count is not aligned to four elements, and the SIMD packing efficiency is lower.

However, in Cifar-10_high, the second and third convolution layers have channel counts aligned to four elements. Therefore, D2I achieves a larger overall execution-time reduction of 15.3%, similar to MNIST_middle and MNIST_high. Note that MNIST and Cifar-10 do not have up to three convolutional layers, so the increase in memory usage due to the D2I algorithm is not significant compared to the decrease in runtime, which ranges from a minimum of 0.032 kB to a maximum of 3.552 kB.

SqueezeNet was tested in the forms of SqueezeNet_1, SqueezeNet_2, and SqueezeNet_3 with a different number of filters and convolution layers. SqueezeNet uses a different table generation algorithm than MNIST and Cifar-10 because it uses padding in all convolutional layers. The results of the SqueezeNet model are shown in Figure 7 and Table 2, and we can see that SqueezeNet_3 performs similarly to MNIST_high and Cifar-10_high with 16.3%, but D2I performance is lower for SqueezeNet_1 and SqueezeNet_2, which have fewer convolution layers and filters than SqueezeNet_3.

This result is mainly caused by the additional padding-aware table generation and zero-filling overhead. In SqueezeNet_1 and SqueezeNet_2, the number of convolution operations is relatively small, so the overhead of preparing padding-aware input references occupies a larger fraction of the total execution time. In SqueezeNet_3, the larger number of convolution layers and filters increases the amount of SIMD computation, making the reduction in Im2col data-copying overhead more beneficial. In these cases, it is actually

possible to perform faster operations when not using an algorithm that omits zeros. The SqueezeNet model has a relatively large memory footprint due to the large number of convolution layers compared to the MNIST and Cifar-10 models, and the heaviest model, the SqueezeNet_3 model, increased memory usage by 11.52 kB.

5.3. Comparison with Direct Convolution Baselines

To further evaluate D2I against convolution methods that do not rely on Im2col, we implemented two additional baselines: Direct convolution and SIMD-optimized Direct convolution. Direct convolution computes each output element by multiplying the input data and filter values directly, without generating an Im2col buffer. The SIMD-optimized version applies SIMD operations to exploit parallelism on the Cortex-M4 processor. The Im2col baseline follows the CMSIS-NN-style implementation using a partial-Im2col buffer and a matrix multiplication micro-kernel.

As shown in Figure 8, D2I achieves the shortest total inference time among the evaluated implementations. Compared with Direct convolution and SIMD-optimized Direct convolution, D2I reduces total inference time by 66.0% and 38.3%, respectively. It also reduces total inference time by 10.3% compared with Im2col. These results show that D2I is faster than the conventional Im2col-based implementation and more effective than Direct convolution baselines on the target Cortex-M4 microcontroller. This is because D2I removes Im2col input-copying overhead while preserving efficient SIMD execution through table-based input referencing.

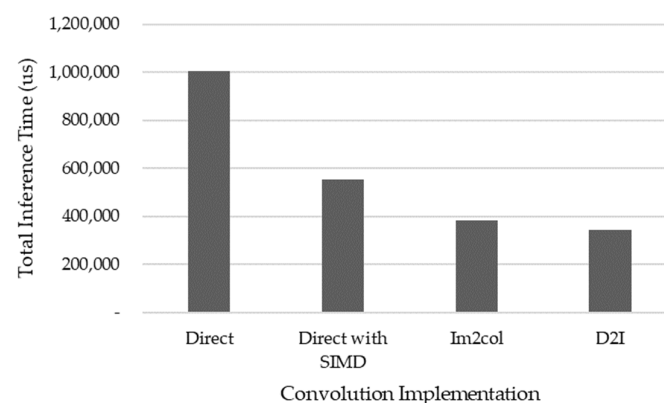


Figure 8. Total inference time comparison of Direct, Direct with SIMD, Im2col, and D2I implementations on the SqueezeNet_3 workload.

5.4. Cross-Framework Performance Evaluation of D2I

To verify the cross-framework applicability of D2I, another framework, NNOM, which uses CMSIS-NN, was tested on the SqueezeNet_3 model on the STM32F746G-DISCO board. As shown in Table 3, D2I reduced the inference time by 14.5% on the STM32F746G-DISCO board, compared with 16.3% on the Arduino Nano 33 BLE Sense. This result shows that D2I can also improve inference performance in an NNOM-based Cortex-M7 environment.

Table 3. Inference time enhancement for TFLM and NNOM.

Model	Board	Reduced Execution Time (%)	Memory Usage (kB)
SqueezeNet_3	Arduino Nano 33 BLE	16.3	11.52
SqueezeNet_3	STM32F746G-DISCO	14.5	11.52

5.5. Limitations and Future Directions

Although D2I reduces repeated Im2col data copying, its performance gain depends on the balance between memory-copy overhead, table-indexing overhead, and SIMD MAC operations. When the number of filters and input channels becomes large, SIMD MAC operations can dominate the execution time, reducing the relative benefit of data-copy reduction. D2I can also be applied to layers whose input channel count is not a multiple of four, but the performance gain may be smaller because SIMD packing and table referencing become less regular.

The current evaluation focuses on TinyML-scale CNN models that can be executed within the memory constraints of Cortex-M microcontrollers. The main target platform, Arduino Nano 33 BLE Sense, uses the nRF52840 microcontroller with 1 MB flash memory and 256 kB SRAM. These resources limit the size of models that can be deployed directly on the board. For example, MobileNetV1-1.0-224 has approximately 4.222 million parameters and requires at least about 4.22 MB for int8 weights alone, while the standard SqueezeNet v1.1 model is about 4.8 MB. These models exceed the 1 MB flash memory of the target board even before considering activation buffers, runtime memory, program code, and the additional D2I table memory.

D2I is a convolution-layer-level optimization and is not restricted to a specific CNN architecture. However, applying it to larger or more recent architectures such as MCUNet, TinyNAS, EfficientNet-Lite, and MobileNetV3 requires model-specific memory-budget analysis, deployment adaptation, and, in many cases, model reduction techniques such as quantization, pruning, layer reduction, or partial workload evaluation. Future work will investigate optimizations that are less sensitive to the number of filters, reduce table-indexing overhead, and evaluate D2I on a wider range of CNN architectures and microcontroller platforms.

6. Conclusions

In this paper, we proposed D2I, a data-copy-reduction-based optimization method for accelerating convolution operations in TinyML frameworks for microcontrollers. We evaluated D2I using MNIST, CIFAR-10, and SqueezeNet models. Compared with Im2col, D2I reduced the inference time by 16.5%, 15.3%, and 16.3% for the heaviest evaluated models, MNIST_high, CIFAR-10_high, and SqueezeNet_3, respectively.

We also compared D2I with Direct convolution and SIMD-optimized Direct convolution baselines on SqueezeNet_3. D2I reduced the total inference time by 66.0% and 38.3% compared with these baselines, respectively, and achieved an additional 10.3% reduction compared with the Im2col implementation. To verify cross-framework and cross-board applicability, we evaluated D2I on the STM32F746G-DISCO board using the NNOM framework. For SqueezeNet_3, D2I reduced the execution time by 16.3% on TFLM and 14.5% on NNOM.

Future work will focus on reducing the table-indexing overhead of D2I, improving its effectiveness in layers dominated by SIMD MAC operations, and further investigating the trade-off between memory usage and inference time in TinyML runtime optimization.

Author Contributions: Conceptualization, Y.-J.K.; software, C.-K.L. and S.-R.O.; validation, Y.-J.K.; investigation, C.-K.L., S.-R.O. and Y.-J.K.; writing—original draft preparation, C.-K.L. and S.-R.O.; writing—review and editing, Y.-J.K.; visualization, C.-K.L. and S.-R.O.; supervision, Y.-J.K.; funding acquisition, Y.-J.K. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) through the Information Technology Research Center (ITRC) support program, funded by the Korean government (MSIT) (IITP-2026-RS-2021-II212051), and by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2025-23523557).

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Strubell, E.; Ganesh, A.; McCallum, A. Energy and policy considerations for modern deep learning research. *Proc. AAAI Conf. Artif. Intell.* **2020**, *34*, 13693–13696. [CrossRef]
2. Cai, H.; Lin, J.; Lin, Y.; Liu, Z.; Tang, H.; Wang, H.; Zhu, L.; Han, S. Enable deep learning on mobile devices: Methods, systems, and applications. *ACM Trans. Des. Autom. Electron. Syst.* **2022**, *27*, 20. [CrossRef]
3. TinyML Market. Available online: <https://www.nextmsc.com/report/tinyml-market> (accessed on 18 May 2026).
4. David, R.; Duke, J.; Jain, A.; Janapa Reddi, V.; Jeffries, N.; Li, J.; Kreeger, N.; Nappier, I.; Natraj, M.; Regev, S.; et al. TensorFlow Lite Micro: Embedded machine learning on TinyML systems. *Proc. Mach. Learn. Syst.* **2021**, *3*, 800–811.
5. Lai, L.; Suda, N.; Chandra, V. CMSIS-NN: Efficient neural network kernels for Arm Cortex-M CPUs. *arXiv* **2018**, arXiv:1801.06601.
6. NNom: Neural Network on Microcontroller. Available online: <https://github.com/majianjia/nnom> (accessed on 18 May 2026).
7. Treveil, M.; Omont, N.; Stenac, C.; Lefevre, K.; Phan, D.; Zentici, J.; Lavoillotte, A.; Miyazaki, M.; Heidmann, L. *Introducing MLOps: How to Scale Machine Learning in the Enterprise*; O'Reilly Media: Sebastopol, CA, USA, 2020.
8. Antonini, M.; Pincheira, M.; Vecchio, M.; Antonelli, F. Tiny-MLOps: A framework for orchestrating ML applications at the far edge of IoT systems. In Proceedings of the 2022 IEEE International Conference on Evolving and Adaptive Intelligent Systems (EAIS), Larnaca, Cyprus, 25–26 May 2022; pp. 1–8.
9. Wang, Y.; Wang, J.; Zhang, W.; Zhan, Y.; Guo, S.; Zheng, Q.; Wang, X. A survey on deploying mobile deep learning applications: A systemic and technical perspective. *Digit. Commun. Netw.* **2022**, *8*, 1–17. [CrossRef]
10. Halford, G.S.; Depoian, A.C.; Bailey, C.P. Machine learning optimization and validation with ST-Microelectronics X-Cube-AI. In Proceedings of the 2023 IEEE MetroCon, Dallas, TX, USA, 2 November 2023; pp. 1–3. [CrossRef]
11. Cheng, J.; Wang, P.; Li, G.; Hu, Q.; Lu, H. Recent Advances in Efficient Computation of Deep Convolutional Neural Networks. *Front. Inf. Technol. Electron. Eng.* **2018**, *19*, 64–77. [CrossRef]
12. Hanson, S.J.; Pratt, L.Y. Comparing biases for minimal network construction with back-propagation. In *Advances in Neural Information Processing Systems 1*; Touretzky, D.S., Ed.; Morgan Kaufmann: San Mateo, CA, USA, 1989; pp. 177–185.
13. Gong, Y.; Liu, L.; Yang, M.; Bourdev, L. Compressing deep convolutional networks using vector quantization. *arXiv* **2014**, arXiv:1412.6115.
14. Denton, E.L.; Zaremba, W.; Bruna, J.; LeCun, Y.; Fergus, R. Exploiting linear structure within convolutional networks for efficient evaluation. *Adv. Neural Inf. Process. Syst.* **2014**, *27*, 1269–1277.
15. Chollet, F. Xception: Deep learning with depthwise separable convolutions. In Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 21–26 July 2017; pp. 1800–1807.
16. Wang, P.; Yang, W.; Fang, J.; Dong, D.; Huang, C.; Zhang, P.; Tang, T.; Wang, Z. Optimizing direct convolutions on ARM multi-cores. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'23), Denver, CO, USA, 12–17 November 2023; pp. 1–13.
17. Dukhan, M. The indirect convolution algorithm. *arXiv* **2019**, arXiv:1907.02129.
18. Park, J.; Bin, K.; Lee, K. mGEMM: Low-latency convolution with minimal memory overhead optimized for mobile devices. In Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22), Portland, OR, USA, 27 June–1 July 2022; pp. 222–234.
19. Hinton, G.; Vinyals, O.; Dean, J. Distilling the knowledge in a neural network. *arXiv* **2015**, arXiv:1503.02531.
20. TensorFlow Lite. Available online: <https://developers.google.com/edge/litert> (accessed on 18 May 2026).
21. Recht, B.; Roelofs, R.; Schmidt, L.; Shankar, V. Do CIFAR-10 classifiers generalize to CIFAR-10? *arXiv* **2018**, arXiv:1806.00451.
22. Iandola, F.N.; Han, S.; Moskewicz, M.W.; Ashraf, K.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv* **2016**, arXiv:1602.07360.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.